Accelerated SQLite Database using GPUs

Glen Hordemann† Jong Kwan Lee‡ Andries H. Smith‡

†Dept. of Computer Science and Engineering
Texas A&M Univ.
College Station, TX 77843
hordemann@tamu.edu

‡Dept. of Computer Science
Bowling Green State Univ.
Bowling Green, OH 43403
{leej, smithah}@bgsu.edu

ABSTRACT

This paper introduces the development of a new GPU-based database to accelerate data retrieval. The main goal is to explore new ways of handling complex data types and managing data and workloads in massively parallel databases. This paper presents three novel innovations to create an efficient virtual database engine that executes the majority of database operations directly on the GPU. The GPU database executes a subset of SQLite's SELECT queries, which are typically the most computationally expensive operations in a transactional database. This database engine extends existing research by exploring methods of table caching on the GPU, handling irregular and complex data types, and executing multiple table joins and managing the resulting workload on the GPU. The GPU database discussed in this paper is implemented on a consumer grade GPU to demonstrate the high-performance computing benefits of relatively inexpensive hardware. Advances are compared both to existing CPU standards and to alternate implementations of the GPU database.

Keywords

Database, SQLite, GPU processing, CUDA

1 INTRODUCTION

The general purpose processing of graphics processing units (GPUs) has begun to transform computing. GPUs provide the ability to perform thousands of operations on data at once, providing supercomputer like power in a single package. This capability empowers developers to greatly increase the performance of existing systems that operate in serial order. The power of GPU programming has been exploited in multi-node computer systems in high-performance computing. Eight of the top fifty supercomputers in the world [16] contain GPUs to increase performance. Research in applying GPUs to solve parallel problems has been done for numerous applications; graphics [8], hydrodynamic solvers [18], differential evolution [17], etc.

A critical business application where the computational power of the GPU can provide significant benefit is database systems. A database system performs a significant amount of repeated calculations on different data. This can occur in table joins or in conditional statements. Both of these database system functions can require significant computation time, slowing system performance and providing an opportunity for GPU programming to offer significant advantages. Traditional databases perform the steps of a query sequentially using the CPU. It is possible to achieve some parallelism by using multiple cores in the CPUs. CPUs have few cores, so attempting to exploit greater parallelism requires the addition of costly additional CPUs, and communication between large numbers of CPUs is problematic; the developer must use some mechanism to regulate the passing of data between CPU memory spaces and the rate at which data is transferred is determined by either inexpensive but slow hardware, such as Ethernet connections, or fast but expensive hardware, such as Cray's Gemini interconnects. A GPU database is feasible due to the considerable inherent parallelism in the way databases operate. A database query consists of a set of operations that are performed on the rows of one or more tables. Each of these operations is repeated, potentially once for each row in the table. Within the query, the execution of these operations is largely data independent; the execution of operations on one row usually does not affect the operations being executed on another row. This exposes a large amount of data parallelism. Each set of operations could be executed simultaneously on every row.

In this paper, we focus on three critical areas for developing a database system on a GPU: data caching to manage the data on the GPU, processing table joins and managing the resultant workload on the GPU, and handling irregular and complex data types. These

 ISBN 978-80-86943-71-8

three areas comprise the fundamental remaining issues with implementing databases on a GPU and addressing them is critical to a successful and high-performance database implementation. The solutions to these problems are presented in an implementation of the SQLite database using a consumer grade GPU. Choosing to use a common, open-source database like SQLite ensures that the solution will be accessible to future developers and will provide real world functionality. Utilizing a consumer grade GPU demonstrates that high-performance computing can be done at a very low price. SQLite supports the SQL standard, which further enhances the accessibility of the solution to future developers. The system described in this paper supports complex SQL options such as querying multiple tables, processing irregular data such as strings, and caching tables using a novel caching scheme for use on the GPU. This designed system is integrated into an application that provides data mining and visualizations of Digital Humanities data as a proof of concept. Here, we note that our work can also be utilized by other work (e.g., pattern recognition, virtual reality, visualization, etc.) for efficient data processing.

Previous research into GPU databases has used data processing primitives in some cases, and simplified implementations of SQL in others. This paper introduces a novel caching system to address the issue of data transfer, providing a solution to the issues of data transfer times and limited GPU memory space. This paper also introduces a scheme for managing the workload of processing data from many tables without overwhelming the GPU. It examines a method for throttling the workload assigned to the GPU based on dividing the required pool of threads into chunks to avoid overloading the GPU. This paper adds the ability to do complex joins, which are critical to the utility of a database in large real-world applications. This paper also investigates and demonstrates the use of strings and complex data structures, where other work has focused on fixed-length values. The fixed length values are excellent for certain data, such as map data that contains coordinate values, but do not provide the utility and flexibility needed for more complex data.

The remainder of this paper is organized as follows. Section 2 provides background information on GPU architecture and SQLite. Related work is discussed in Section 3. Section 4 provides details of the new GPU-accelerated database engine. Section 5 gives the experimental results for the improvements made in this paper. Section 6 concludes this paper.

2 BACKGROUND

In this section, the architecture of the GPU and the inner workings of the SQLite are briefly discussed.

2.1 GPU Architecture

A GPU contains hundreds of individual cores organized into streaming multiprocessors (SMX); e.g., NVIDIA 680 GPU has 1536 cores organized into 8 streaming multiprocessors with 192 cores each. The GPU executes functions, called kernels. These kernels are launched by the CPU, using a programmer-specified number of thread blocks, which contain a specified number of threads per block. These thread blocks and their threads are divided into 32-process warps and scheduled on the streaming multiprocessors as free cores become available. Global memory on the GPU is of much higher latency than typical GPU memory. The GPU masks this latency by context switching from warps whose threads are waiting for a memory access to warps that have the required data currently available. This is made possible by extremely low latency context switching, which allows the process scheduler to swap the warps running on a streaming multiprocessor. This functionality assumes that a significantly greater number of processes than cores are being executed on the GPU, so that there are always ready warps to switch to in order to hide memory latency.

In addition to the GPU having a different design than the CPU, the GPU is a separate ecosystem from the CPU and contains GPU-specific memory, scheduler, and processors. Data must be explicitly moved to the GPU, and the GPU can only access data stored in the GPU memory. Conversely, the CPU cannot directly access the GPU memory. In order to access CPU resources, the GPU must work through the CPU. This separation of systems requires extra steps for translation, instruction, and control.

The limitations on CPU and GPU interaction impose special complications for database processing on the GPU. The distinct memory spaces of the two systems require mechanisms for translating data structures. In the case of databases, this requires both the transformation of irregular data such as strings and the processing of the table data structure. The smaller size of GPU memory also requires a system for managing the data that is transferred to and retained on the GPU.

2.2 SQLite

SQLite is an open-source database system that is widely used for embedding a database in applications on computers, smartphones, and tablets. Unlike other databases such as Oracle, SQLite can be compiled directly into the source code and run as part of a program instead of being called as a service that runs as a separate process. SQLite is public domain software and is one of the most commonly used database systems in the world. SQLite supports the standard SQL syntax. SQL is an easy-to-use language that is ubiquitous in industry. It is commonly used not just by programmers

but by business professionals of all types. Supporting such a common interface ensures that the barrier to entry to use the GPU database system is trivial.

SQLite allows the use of complex logic in queries of the database, allowing disparate tables to be joined together by common values and a selection of results to be chosen based on user-defined criteria. The joins are of critical interest, as the process of finding matching values between rows of data in two tables can be very repetitive and computationally expensive. Likewise, the task of finding rows that meet certain criterion is also a repetitive and computationally taxing task. Both of these tasks are ideal for GPU assistance.

SQLite processes queries by converting the SQL query into an opcode program that runs on a virtual machine, not unlike the way Java programs are transformed to run on the Java Virtual Machine. Each SQLite opcode can have up to five parameters which provide additional information needed for the opcode to execute on the virtual machine, which is referred to as the Virtual Database Engine (VDBE). The output from the virtual machine is returned as rows of data to the calling program. The standard Virtual Database Engine, as a key aspect of the database system, was replaced with a virtual engine designed to run on the GPU.

3 RELATED WORK

This paper advances prior work done in the field of high-performance, massively-parallel databases. Some prior work focused on fixed length data types such as integers and doubles, often coupled with straightforward single table queries. Other work focused on using primitives that are not a component of standard SQL databases. In this section, some of the key work in the research area are briefly discussed.

As mentioned previously, the area of GPU databases has been a growing area of research. At one end of the spectrum is work in adding bolt-on GPU database modifications. Work with PostgreSQL [7] has explored this possibility. External procedures, such as those developed by Bandi et al. [4] provide examples of offloading processing work to GPU functions. Proof-of-concept work has been done to examine specific aspects of massively parallel databases, without a full database implementation behind it. Relational join [12] and efficient sorting algorithms [13, 14, 15, 5, 10] are two areas where proof of concept work has been done. Databases that do the full processing of the query on the GPU have been researched by Bakkum et al [3] in their work on SQLite databases, which with the work by Chang et all [6] served as an inspiration for this paper. This paper extends their work by adding caching, more complex join and query operations, demonstrating a method for handling workload management, and handling irregular data. Methods of handling complex data were addressed by Bakkum et al. [2] using an approach they refer to as Tablets. Tablets collect groups of rows into a single tablet, and coalesce the rows into a column-oriented format. Thus, while all of a row will appear in a single tablet, the entire column will not. We note that our work takes a different approach, collecting the entire column into a single data structure in order to optimize operations such as joins and conditional that match to a specific single column. The cost of data transfer, which motivates our caching system, was explored in a paper by Gregg and Hazelwood [11]. A GPU database that exists entirely in memory and never leaves the GPU [9, 1] can alleviate this data transfer issues, but is limited in scope.

4 NEW GPU-ACCELERATED SQLITE

This paper introduces three novel innovations to create an efficient virtual database engine that executes the majority of database operations directly on the GPU. The GPU database executes a subset of SQLite's SELECT queries, which are typically the most computationally expensive operations in a transactional database.

4.1 New Design

Three new improvements are examined in this database system; a caching strategy to handle the storage of tables in GPU memory, a method for handling irregular data and complex data structures on the GPU, and a process for handling complex joins on the GPU and managing the resultant workload on the GPU.

4.1.1 Caching Strategy

Data storage in this design is handled at three different levels. The lowest and slowest level is the hard drive, which is the long-term storage for the database. Above this is the CPU memory space, into which tables are loaded when they are needed and retained for future use. The top level is the GPU main memory, which

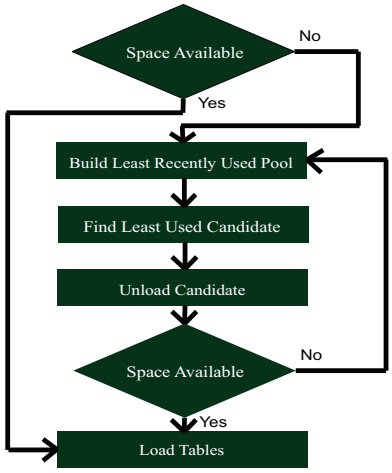

Figure 1: Cache Replacement Policy

 ISBN 978-80-86943-71-8

holds the tables being used in the current query, with any remaining space used to retain tables used in previous queries. This top level contains the catching strategies that are of interest to this paper.

There are a number of variables that influence the catching scheme that are known in advance. The size of any given table is known in advance and the sizes of the tables are non-uniform. The relatively small number of tables in a database means that the usage for all tables can be tracked, establishing which tables are more frequently used and which are more recently used. The cache replacement scheme implemented in this system takes advantage of this knowledge as shown in Fig. 1. When more space is needed on the GPU for a query, the system selects a table for replacement by scanning the tables on the GPU and building a pool of candidates based on which tables on the GPU have been least recently used. From this pool, the candidate with the least number of uses is selected for replacement. Space availability is then checked again. If additional space is still required, a new pool of candidates is selected and the process is repeated. This continues until enough space is available on the GPU for the tables.

An assumption has been made in the current design of this system. That assumption is that the tables required for the query will fit into the memory space available on the GPU. (Modifying the database to handle tiling of tables to encompass queries too large for the GPU memory is an area for future work.)

4.1.2 Irregular Data and Complex Data Objects

Processing irregular data, or data whose size is not constant from one instance to the next, requires special solutions to overcome specific obstacles that the uncertainty of the data size creates. The issues of arrays of separate memory allocations must be resolved by coalescing these arrays into single memory allocations. The issue of transferring complex data objects assembling disparate metadata information is resolved by creating special CUDA functions to assemble data on the GPU side. Finally, a solution to the coalescing of result data must be created that avoids race conditions.

Coalescing Irregular Data: The issue of arrays of separate memory allocations most often comes in the form of strings. A string is not a discrete object in C/C++, but rather is a pointer to an array of characters. Creating an array of strings results in creating a pointer to an array of other pointers to arrays. This is problematic because memory locations in the CPU memory space have no relation to memory locations in the GPU memory space. Passing an array of pointers simply passes an array of gibberish as far as the GPU memory space is concerned.

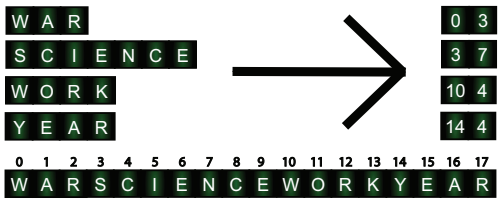

Figure 2: Coalescing Strings into a Superstring

The solution is to coalesce all these different arrays into a single array, as shown in Fig. 2. The individual arrays are retained as separate virtual entities by providing starting indexes and lengths of each individual array within the coalesced array. The number of arrays needed transforms from an N arrays to two arrays; one for the array offsets and lengths, which has a length of 2N, and one which contains all the values of the separate arrays concatenated together which has a length equal to the sum of the lengths of the N arrays.

While it is possible to create many individual arrays and pass each one individually, the overhead of numerous data copies from the CPU memory space to the GPU memory space rapidly degrades performance. Coalesced arrays reduce this overhead from *O(n)* to *O(1)*.

Complex Data Structures: The multiple objects copied over to the GPU must be assembled into a single complex object; the arrays of database values must be reassembled into table objects that are then reassembled into the database object. This assembly must take place on the GPU because only the GPU can interpret or access GPU memory spaces. The pointers to those memory spaces are currently only known to data objects on the CPU, yet only GPU functions can manipulate the GPU memory spaces they refer to.

The solution to this issue is to create linking functions that attach one data object to another. For example, the CPU instructs the GPU to make a *database* object. This object contains pointers to table structures. Initially these pointers do not point to anything. The CPU then instructs the GPU to create a *table* object. The CPU has no mechanism to tell the *database* object's table pointers to point to the *table* object. This issue is resolved by the creation of the linker function. A linker function accepts two pointers as arguments and attaches the second to a specified pointer contained in the object the first pointer refers to.

The process for loading a table is shown in Fig. 3. The table object is allocated on the GPU and initialized. Then a *column types* array is created. Next, a linker is called to attach the array to the *table* object. The process then begins for each column. The columns of the table are processed sequentially. For each column, the *data value* array is allocated on the CPU, copied to the CPU, and then a linker function is called to attach it to the *table* object. For string data, which is of varying length, the "value" contains the starting index of each sub-array, or string of characters, in the coalesced array, which is called a superstring. For this string data,

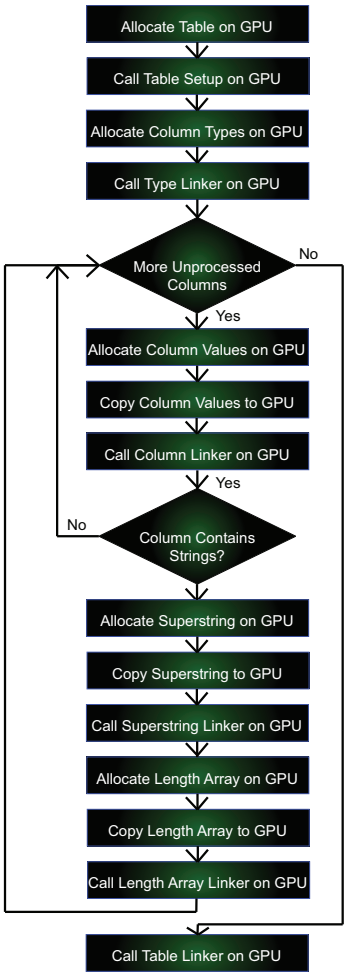

Figure 3: Table Load Process

each superstring is allocated, copied, and attached by a linker, and then the array of sub-array lengths is also allocated, copied, and attached by a linker function. Finally, the *table* object itself is connected to the *database* object. This means that a single reference can be passed to refer to the database, instead of a large, arbitrary, and potentially unworkable number of references that point to the assorted columns of all the tables in the database.

Coalescing Results: With varying sized input, the output is also potentially of any size, both in number of results and in the size of each result entry itself. This introduces two levels of uncertainty, and it is the second level of complexity that requires a new solution.

The traditional way of handling result sets is not robust enough to handle varying length data. The traditional way to coalesce result sets is to allocate an array of result elements for the results and have each thread claim a slot in the array as needed. This handles one level of uncertainty, the number of result elements, but relies on the fixed length of each result set to calculate the start point of any particular slot. Without knowing the lengths of elements, it is impossible to find a slot in the result set until all prior result sets have been calculated.

```
case OP_ResultRow
{
  int ResultSize = 0;
  // d_ResultSize is the global result size

  // Calculate string length
  for(int t=StartRegister; t<=StartRegister+
      RegisterCount; t++)
    ResultSize += RegisterSize;

  // Get End point and update global result
      pointer
  int StartPoint = atomicAdd(d_sizeResults,
      ResultSize);

  // Collect result set.
  for(int t=StartRegister; t<=StartRegister+
      RegisterCount; t++)
    ResultSet += RegisterContents+Separator;

  // Place result set into result collection
      at StartPoint
  ResultCopy(ResultCollection, StartPoint,
      ResultSize, ResultSet);

  // Update the count of total result entries
  atomicAdd(d_numResults,1);
  break;
}
```

Figure 4: Atomic Coalescing of Results

The solution to this problem requires atomically claiming space in the result set. Each thread calls an atomic adding function that adds the length of the result set it contains to a global variable. This function also returns the starting value of the variable before the addition takes place. This allows each thread to claim a unique portion of the result set, using the initial value of the global variable as the starting point and the length of the result set it is processing as the end point. Pseudocode for this process, as contained in the *ResultRow* opcode, is shown in Fig. 4.

4.1.3 Workload Management and Complex Joins

The process of joining multiple tables creates a multiplicative increase in the number of permutations that must be evaluated. Each row in a table must be compared to every row in the joined table in order to connect that data in the tables together correctly. Good SQL design and the addition of indexes can minimize the number of comparisons that need to be done in many cases, but there will always be instances where a table join must be done in the simple, brute force manner. In all circumstances where the number of permutations to be evaluated exceeds the capacity of the GPU to process the entire set at once the workload assigned to the GPU must be managed.

The workload management is also influenced by contention for memory resources. There may be cases where memory access is the constraining factor and the

 ISBN 978-80-86943-71-8

Line	Opcode	P1	P2	P3
7	Rewind	0	22	0
8	Rewind	1	22	0
9	Rewind	2	22	0
10	Column	0	0	1
11	Column	1	0	2
12	Column	2	0	3
13	Lt	1	15	3
14	Le	2	19	3
15	Column	0	0	5
16	Column	1	0	6
17	Column	2	3	7
18	ResultRow	5	3	0
19	Next	2	10	0
20	Next	1	9	0
21	Next	0	8	0
22	Close	0	0	0
23	Close	1	0	0
24	Close	2	0	0
25	Halt	0	0	0

Table 1: Join of Three Tables

GPU is capable of managing more threads computationally. In such cases, the assigned workload to the GPU must be throttled to avoid resource starvation.

This need for workload management most often occurs when joins of multiple tables are done, using multiple criteria. The more tables joined, the more complex work must be done. It is not sufficient to simply assign one thread to each table permutation. An example of such a section of a SQLite program where the number of combinations requires workload management is shown in Table 1. In this table, the presence of three Next opcodes indicates that there are three levels of nested loops, each one of which executes the inner loop multiple times. The number of iterations of the inner loop is equal to as much as the number of rows in the inner table times the number of rows in the middle tale, times the number of rows in the outer table. If each table has only a thousand rows, then the innermost instructions are executed a billion times.

The solution to managing this vast amount of processing is to divide the workload into chunks. Each chunk consists of a grid of threads. How data is assigned to threads in a chunk is critical due to the need to maximize the impact of indexing, and minimize memory accesses and individual thread processing time.

The assignment of data is begun by dividing tables into two groups. The code for this division is shown in Fig. 5. Tables that are not indexed, which are ephemeral, are selected to belong to the static group that have each row in the table assigned to a separate thread. Tables are selected without regard to the total number of threads allowed in a grid or the number of rows they contain. All tables that are not assigned

```
// Get all row sizes of tables we are using.
for(int t=0; t<numTables; t++)
  tableRows[t] =
    dbProfile.tableInfo[tableMeta[t].
        indexOnCPU].rows;

int blocksize = SQLCUDA_BLOCKSIZE;
int tableIndex = 0;

//Skip to first non-Ephemeral table.
while(tableMeta[tableIndex].ephemeral)
    tableIndex++;

int gridsize=(int)ceil(tableRows[tableIndex]
    /(double)blocksize);

tableIndex++; // Skip to next table. The
    first one is already handled.

// Process all the non-Ephemeral tables left.
for(int t=tableIndex; t < tableCount; t++)
{
  if( !(tableMeta[t].ephemeral) )
  {
    if(gridsize < 2000000 && (gridsize *
        tableRows[t]) < 2000000)
      gridsize*=tableRows[t];
    else tableRows[t] = 0-tableRows[t];
  }
  else tableRows[t] = 0-tableRows[t];
}
```

Figure 5: Division into Processing Groups

to the static group are assigned to the iterate group. Programmatically, the static group is indicated by a positive number of rows and the iterate group is indicated by the row count being negative. Tables in the static group are not iterated through; a single thread processes a single row and only iterates through tables in the iterate group. If there are no tables in the static group, a table from the iterate group is promoted to the static group. In the database engine, a boolean value is set based on whether the row count is positive or negative and stored in the moveCursor variable for the cursor. This boolean is then used by the Next opcode (shown in Fig. 6) to determine whether the virtual database engine should advance the cursor a row and jump the program counter to the indicated line.

The second step is the assignment of the static group to chunks. The size of the grid in each chunk is dependent on the number of nested loops in the opcode program. More nested loops are more likely to result in slower execution, so a correspondingly lower number of threads need to be assigned to the GPU in each chunk. Each row of a non-indexed static group table is assigned a thread in a chunk. This results in a single thread only executing for a single unique combination of rows from each of the static tables. The row to be executed in each thread is calculated from the threads block, grid, and chunk numbers. Chunks are then executed sequentially, which means that threads assigned

```
case OP_Next:
{
  int p1=currentOp->p1;

  if(moveCursor[p1]&&(cursorRows[p1]+row[p1]
      == -1))
  {
    pc = currentOp->p2-1;  // Set program
        counter to line P2.
    row[p1]++; // Move cursor to next row in
        table.
  }
  break;
}
```

Figure 6: Next Opcode

to the same warp, block, or grid will be accessing the most spatially local rows in a table.

5 EXPERIMENTAL RESULTS

Several different techniques were used to evaluate the performance of these improvements. A database querying and caching simulator was built to compare different caching strategies. The performance of the GPU database was compared with GPU-level caching turned on and turned off. Queries of increasing complexity were run to compare cached GPU performance to standard SQLite cached CPU performance. Irregular data performance was compared to regular data performance, using strings and integers respectively, and database engine execution time was measured for returning irregular and regular data using the same method. Finally, cached GPU performance was compared to standard SQL CPU performance for several complex SQL statements.

The test machine used in the caching tests is a Pentium i7 920 running at 2.67 GHz with an NVIDIA GeForce GTX 660Ti with 2 GB of memory. This GPU has 1344 cores arranged into 7 streaming multiprocessors. GPU to CPU comparisons were performed on a Pentium i7-3920k running at 3.2 GHz with 64 GB of memory and an NVIDIA GeForce GTX680 GPU. This GPU has 1536 compute cores, arranged into 8 streaming multiprocessors, and 2 GB of memory. The database was stored on an Intel 240GB SSD. Both systems run Windows 7 and both GPUs support Compute Capability 3.0. The database program was written using CUDA version 5.0 and C++ with Microsoft Visual Studio.

5.1 Testings on Caching Strategy

The caching strategies used in this system were tested in two different ways. To determine an optimal cache replacement algorithm, a caching simulator was built to simulate a wide variety of database configurations, drawing on past experience as a database administrator. Once a caching scheme was chosen, the impact of that caching scheme on performance was determined.

Strategy	Hit Count	Hit Bytes
Random	79.3	79.7
LRU	79.9	80.6
Least Used	78.6	83.1
Biggest First	84.4	81.0
Biggest of Least Used	82.9	77.2
Least Used of LRU	82.4	84.3
LRU of Least Used	78.7	79.2
Biggest of LRU	82.8	76.8

Table 2: Caching Strategies (Pool Size: Six)

Caching Strategy Selection: Table 2 shows the results of one run of the cache simulator. Various databases were modeled. Different database configurations cause an alteration in the percentages of hits, “Least Used of LRU" model was typically the most efficient model. A variety of pool sizes were evaluated, and a pool size of six was consistently the best performing. The “Hit Bytes" is the most critical number, as the cost of transferring table data is the dominant factor in our caching benefits, not the number of tables in the cache.

It should be noted that database systems can be modeled that result in strategies other than “Least Used of LRU" being the most optimal. These test models were in the minority of the test cases generated and required significant tweaking of the simulator parameters before this behavior was present.

Caching Strategy Performance: The performance of the caching system in overall performance is shown in Table 4. This table shows the change in query execution time when a table is not cached on the GPU compared to when it is cached on the GPU. The more frequently a cache hit occurs, the greater the speedup of the query due to caching. The table shows a baseline 100% hit rate and can be scaled from there.

The greatest performance increase comes from single table queries. The first query demonstrates this effect. In this query, there is little additional computation done to slow down the query, either in joins or in conditional statements. Additionally, there is no optimization done to minimize data access by the database system which would add overhead.

The second query demonstrates the impact of optimizations. The Publication table is much smaller than the Document table; thus, it provides only a small impact to the penalty for no caching. The extra computations required to do the join amortizes the penalty of not caching, reducing the speedup. It is worth noting that the ordering of the SQL statement has an impact on the performance of the SQL, and a different formulation of the same query will result in a slightly slower query.

The final query demonstrates that, even in an instance where a significant amount of computation is occurring, the impact of caching is still felt. A significant

 ISBN 978-80-86943-71-8

amount of optimization is done by the database system in a query this complex, such as the creation of dynamic indexes to replace tables, which will negate the need to load some tables. Even joining five tables, three of which are large, and with a complex conditional clause, the additional of caching provides a speedup of 1.34.

5.2 Testings for Irregular and Complex Data

Irregular Data performance was evaluated in two ways; by comparing string performance of the GPU database against string performance of the CPU database, and by comparing string performance of the GPU database against numeric performance of the GPU database. The first comparison demonstrated that the GPU database exceeded the CPU implementation. The second comparison showed that the string implementation was efficient and did not suffer serious performance degradation due to the use of varying sized data.

Performance comparisons were made between the CPU and GPU databases using two categories of SQL statements. The first category was SQL statements with selection criteria, using both single and compound criteria. The criteria used strings to select the rows to be returned. The third category was SQL statements with joins, using both single and complex joins. These statements used strings both in criteria to select rows and in the return values. The tables used were the Authors and the Document tables. These tables were selected because they contained a large volume of string data and they represented both medium and large sized tables.

The SELECT queries with conditional clauses showed a performance gain by the GPU database over the CPU database. The execution times for these queries are shown in Table 5. The single criteria of Query 1 and Query 2 demonstrated the speedup achieved with string condition clauses. These queries also demonstrated that larger tables, which have more processing to perform, achieved a greater speedup. The multiple conditional clauses of Query 3 and Query 4 demonstrated that a greater speedup was achieved when more conditions were evaluated. This indicates that each string condition statement was faster in the GPU database than in the CPU database, and additional condition statements would result in an even greater speedup. Query 5 showed the performance with multiple joins between the CPU database and the GPU database still reflected a significant speedup.

String performance must be comparable to integer performance in the GPU database. This was evaluated by comparing the performance of similar queries on the GPU database. The conditional clauses of the statements were written with one version using strings to select records and the other version using numeric values on the same table. The results are shown in Table 7 and demonstrated that the string performance of the GPU was similar to the numeric performance. The queries demonstrate that the performance between string and numeric SQL SELECT statements was well within an order of magnitude, and was less than a factor of two in difference.

The performance of coalescing the results when varying length data was present was determined by comparing the performance of fixed data results with varying length data results. This comparison was done with two representative tables; the Authors table and the Document table. One field was selected in each statement, returning either a fixed length value (id) or a varying length value (firstname and title). The execution time of the virtual database engine itself was evaluated, without consideration for other issues such as caching, data transfer time, or table setup.

The performance of the coalesced varying length data matched the performance of the fixed length data. The Authors table showed slightly faster execution time for the varying length data. This was appropriate because there was slightly less data to process when coalescing the results. In the much larger Document table, the VDBE took almost twice as long to run. However, this statement returned five times as much data, which must be coalesced into the result set. The larger result set explains the longer run time, indicating the success of this coalescing method.

5.3 Testings for Complex Joins

Join performance was evaluated based on the performance of the GPU SELECT statements that contained joins. The performance of the query on the GPU database was compared to the performance of the query on the CPU database. Queries that join tables based on indexes and without the benefit of indexes were compared. The results of these queries are displayed in Table 8.

The effectiveness of a query without the use of a dynamically created index was demonstrated in Query 7. This query joins two tables on a text field. The results of this query showed a significant increase in performance of the GPU database over the CPU database. The execution of Query 8 allowed the testing of a large data set where indexes were used to accelerate the join. The query showed significant speedup on the GPU database compared to the CPU database, indicating the success of the join methods used in the GPU database. The results of Query 9 also showed a significant speedup, but the speedup was not as great as the speedup of the smaller join statement. This reflected the relatively large time cost of creating indexes compared to the relatively small cost that execution of that index brings; indexes are extremely fast but making them can be slow. In Query 9 more indexes are created, and this has a small negative effect on the efficiency of the query.

ID	SQL Statements
1	SELECT * FROM Authors WHERE lastname = 'Campbell'
2	SELECT * FROM Document WHERE title = 'Population'
3	SELECT id FROM Authors WHERE fullname = 'Edna F. Campbell' OR firstname = 'Jim'
4	SELECT id FROM Document WHERE title = 'Population' OR pagerange = 'pp. xxxviii-xxxix'
5	SELECT a.id FROM Document d, DocumentAuthors da, Authors a, PubEdition pe, Publication pu WHERE (d.id=da.documentid AND a.id=da.authorid AND pe.id=d.pubeditionid AND pu.id=pe.pubid) AND ((da.documentid=81372 AND d.repositoryID = 1) OR pu.title ='The Scientific Monthly' OR pu.title='Scientific American' OR a.firstname='John' OR d.title='Rural Conditions in the South'
6	SELECT d.title FROM Publication pu, Document d WHERE pu.title=d.title AND d.title='Campbell'
7	SELECT * FROM Document d, Authors a WHERE d.title=a.firstname
8	SELECT d.id FROM Document d, DocumentAuthors da WHERE d.id=da.documentid AND da.documentid=813
9	SELECT a.id FROM Document d, DocumentAuthors da, Authors a, PubEdition pe, Publication pu WHERE (d.id=da.documentid AND a.id=da.authorid AND pe.id=d.pubeditionid AND pu.id=pe.pubid) AND ((da.documentid=81372 AND d.repositoryID = 1) OR pu.id < 2 OR pu.publisherid<2 OR a.id<2 OR d.pubeditionID<2)

Table 3: Samples of Tested SQL Statements

SQL	No Cache (ms)	Cached (ms)	Speedup
2	62.443	1.963	31.8
6	63.185	4.827	11.94
5	225.156	168.259	1.34

Table 4: GPU Caching Performance

SQL	GPU (ms)	CPU (ms)	Speedup
1	6.73	33.61	4.99
2	7.44	298.78	40.16
3	5.01	47.69	9.51
4	8.13	386.92	47.59
5	271.36	8341.45	30.74

Table 5: CPU vs. GPU with Selection Criteria

6 CONCLUSION

In this paper, we have shown several improvements to existing GPU database research. A number of caching strategies to manage data storage on the GPU were explored and the most effective solution was evaluated, and a method for managing the workload on the GPU was presented. A method for coalescing and managing variable length data, both in moving data to the GPU and in retrieving results, was demonstrated and shown to perform almost as well as simpler, fixed-size solutions. A method for moving complex data structure to the GPU was demonstrated. Finally, complex joins, including those involving irregular data, were demonstrated in the GPU database.

Future work in this system will require expanding the database implementation to tile databases that are too large for the system into the GPU memory. Even with the advent of direct GPU memory access in CUDA 6, direct management of the GPU memory spaces will yield much better performance results.

Caching could also be improved by using a predictive pre-caching scheme. This would expand the opportunities to benefit from the performance improvements of caching. In interactive systems, preemptive caching based on user input as SQL statements are being entered could also increase performance by allowing the caching to be done before the SQL statement is executed.

The database can be expanded to further implement SQL features on the GPU. These additional features, such as INSERT, UPDATE, and DELETE statements, which are now handled purely by the CPU, may reveal new challenges when coupled with joins and varying size data types. The ability to add stored procedures to the GPU database, written to run on the GPU, allow user-implemented complexity. This feature is already available in some other systems.

The investigation of alternate methods of accessing GPU's programmatically is another area of future research. CUDA only supports NVIDIA video cards, and ATI cards have a strong reputation for better integer performance than NVIDIA cards. The use of alternate programming models, such as DirectCompute or OpenCL, may also provide altercate solutions to some of the problems that have been addressed specifically with CUDA.

7 REFERENCES

[1] Bakkum P. and Chakradhar, S., High Performance Computing on Graphics Processing Units, NEC Laboratories America, `http://hgpu.org/?p=7180`, 2012.

[2] Bakkum P. and Chakradhar, S., Efficient Data Management for GPU Databases, NEC Laboratories America, `http://pbbakkum.com/virginian/paper.pdf`, 2013.

[3] Bakkum, P., and Skadron, K., Accelerating SQL Database Operations on a GPU with CUDA, in Proc., 3rd Workshop on General-Purpose Computation on Graphics Processing Units, Pittsburgh, Pennsylvania, 2010, pp. 94–103.

SQL Statement Description	Numeric (ms)	String (ms)	Speed up
Authors Table, Single Criteria	3.42	6.73	0.51
Document Table, Single Criteria	6.32	7.44	0.85
Authors Table, Multiple Criteria	3.74	5.01	0.75
Document Table, Multiple Criteria	6.73	8.13	0.83
Complex Join	271.36	298.94	1.10

Table 6: Numeric vs. String Conditional Statement

SQL Statement Description	Bytes Processed	VDBE Execution (ms)
SELECT id FROM Authors	505,380	1.31
SELECT firstname FROM Authors	482,029	1.02
SELECT id FROM Document	2,791,250	6.06
SELECT title FROM Document	14,719,204	11.76

Table 7: Time of Virtual Database Engine Execution

SQL	GPU (ms)	CPU (ms)	Speedup
7	62.85	2737.81	43.52
8	5.83	223.74	38.38
9	278.83	8261.43	29.63

Table 8: Join Performance

[4] Bandi, N., Sun, C., Agrawal, D., and El Abbadi, A., Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations, in Proc., 30th Int'l Conf. on Very Large Data Bases, Toronto, Canada, 2004, pp. 1021–1032.

[5] Cederman, D. and Tsigas, P., GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors, Journal of Experimental Algorithmics, Vol. 14, 2010, pp. 4:1.4–4:1.24.

[6] Chang, Y.-S., Sheu, R.-K., Yuan, S.-M., and Hsu, J.-J., Scaling database performance on GPUs, Information Systems Frontiers, Vol. 14 (4), 2012, pp. 909–924.

[7] Child, T., Parallel Image Search Using PostgreSQL and PgOpenCL Presentation, PG-Con 2011: The PostgreSQL Conf., Ottawa, 2011.

[8] Costa, V., Pereira J.M., and Jorge J.A., Fast Compression of Meshes for GPU Ray-Tracing, in Proc., 21st Int'l Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision, Plzen, Czech Republic, 2013, pp. 10–18.

[9] Ghodsnia, P., An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads, in Proc., 38th International Conference on Very Large Data Bases, Istanbul, 2012, pp. 54–59.

[10] Govindaraju, N., Gray, J., Kumar, R., and Manocha, D., GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management, in Proc., 2006 ACM SIGMOD Int'l Conf. on Management of Data, Chicago, 2006, pp. 325–336.

[11] Gregg, C. and Hazelwood, K., Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer, in Proc., IEEE Int'l Symp. on Performance Analysis of Systems and Software, Austin, Texas, 2011, pp. 134–144.

[12] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P., Relational Joins on Graphics Processors, in Proc., 2008 ACM SIGMOD Int'l Conf. on Management of Data, Vancouver, Canada, 2008, pp. 511–524.

[13] Merrill, D.G. and Grimshaw, A.S., Revisiting Sorting for GPGPU Stream Architectures, in Proc., 19th Int'l Conf. on Parallel Architectures and Compilation Techniques, Vienna, Austria, 2010, pp. 545–546.

[14] Satish, N., Harris, M., and Garland, M., Designing Efficient Sorting Algorithms for Manycore GPUs, in Proc., 2009 IEEE Int'l Symp. on Parallel & Distributed Processing, Rome, 2009, pp. 1-10.

[15] Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., and Dubey, P., Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, in Proc., 2010 ACM SIGMOD Int'l Conf. on Management of Data, Indianapolis, Indiana, 2010, pp. 351–362.

[16] Top 500 Supercomputer Sites, `http://www.top500.org/list/2013/06/`, June, 2013.

[17] Qin, A.K., Raimondo, F., Forbes, F., and Ong, Y.S., An Improved CUDA-based Implementation of Differential Evolution on GPU, in Proc., 14th Int'l Conf.e on Genetic and Evolutionary Computation, Philadelphia, 2012, pp. 991–998.

[18] Westphal, E., Singh, S.P., Huang, C.-C., Gompper, G., and Winkler, R.G., Multiparticle collision dynamics: GPU accelerated particle-based mesoscale hydrodynamic simulations, Computer Physics Communications, Vol. 185 (2), 2014, pp. 495–503.